Distributed Lazy Evaluation: A Big-Step Mechanised Semantics

(Author's Version)

Seyed H. HAERI (Hossein)
Institute for Software Systems,
Hamburg University of Technology, Germany
hossein@tu-harburg.de

Sibylle Schupp
Institute for Software Systems,
Hamburg University of Technology, Germany
schupp@tu-harburg.de

***Abstract*—This paper presents a big-step operational semantics for distributed lazy evaluation. Our semantics is an extension to the famous heap-based semantics of Launchbury for lazy evaluation. The high level of abstraction in our semantics helps us to easily prove different properties that are of interest to task distribution. Most importantly, we give criteria which establish a notion of bisimilarity between heaps. We also prove the validity of an induction principle that is used for proving observational equivalence between programs written in our system. Additionally, we explain how the component-based nature of a formerly presented material for mechanisation of Programming Languages (PLs) facilitates experimentation with the semantics of this paper.**

***Keywords*—Big-Step Operational Semantics, Distributed Programming, Lazy Evaluation, Language Mechanisation**

I. Introduction

Laziness avoids variable evaluation so long as the value of that variable is not needed elsewhere in the expressions of interest. Moreover, as formalised first in [16], laziness prohibits more than one evaluation for every variable through the intuitive concept of heaps. The motivation for this indirection is saving unneeded evaluations. There are various reasons, however, that the programmer may choose to enforce strictness (e.g., controlling execution flow and space or time efficiency). To accommodate that, works such as [24], [5] extend lazy evaluation with selective strictness but for sequential programs.

With the growth of multicore systems, different research threads have tried to study non-sequential ways to enforce strictness. Examples include employing parallelism [9], [2] or distribution [11], [10]. These works, however, all offer small-step semantics that – although much more natural for describing interleaved computations – is not as easy as big-step systems to reason about. More specifically, proving that programs operate similarly is known to be particularly difficult in small-step semantics. (See [11] for some denotational approaches in proving program equivalence for distributed lazy evaluation.)

Proving observational equivalence in the lazy evaluation campaign was first studied in [5]. These proofs were all for selective strictness and for a flavour of laziness that unfortunately suffers from expressiveness restrictions. Later on, it was shown in [6] that the same results hold even when those restrictions are removed. The most notable technique used in the latter works for proving observational equivalence is a new induction principle called *Induction on the Number of Manipulated Bindings* (INMB).

In this work, we provide a new big-step operational semantics for distributed lazy evaluation (Definition 6). Our semantics, although not the first big-step system for this purpose, is remarkably simpler than its single predecessor (i.e., [21]). Thanks to the simplicity of our semantics, we can prove various results about it that are of interest to task distribution: We prove INMB (Theorem 2) that, in return, is used in proving bisimilarity between the heaps used over an evaluation (Theorem 7). We also prove two observational equivalences that dismiss the need for performing certain parts of derivations (Corollary 1 and 2). Likewise, we prove that the order in which certain heaps are merged does not matter (Corollary 4). The two latter results can be viewed as optimisation legislations. Finally, we show that – despite addressing distribution – our semantics enjoys a notion of determinism (Theorem 1). The proofs we omit here can all be found in [7].

A plus about our semantics is that it is mechanised. Mechanisation of a language is implementing it for the experimental study of its characteristics and conduct. One interacts with the mechanisation to discover otherwise inapparent facts or flaws **in action**. We demo our system's mechanisation to explain how the particular mechanisation approach we used caters distribution at a high level of abstraction. We specifically present our distribution rule's mechanisation for its pivotal role.

This paper is organised as follows: We start by presenting our syntax and semantics in Section II. Some properties of our system are explored in Section III. Section IV focuses on heap bisimilarity and its importance for task distribution. Then, in Section V, we discuss the mechanisation of our system. A review on the related work is provided in Section VI. Finally, in Section VII, we conclude and discuss future work.

II. Syntax and Semantics

This section presents the syntax and semantics of our system. We also provide a few related definitions and notations that we will refer to later.

Definition 1: Fix a countably infinite set of **variable symbols**. $x, y, z, \ldots$ and $x_1, x_2, \ldots$ will range over variable symbols. Define **the DLE**[1] **expressions** by

$e ::=$			Exp
	x		Var
	$e\,x$		App
	$e\#x$		Srp
	$\mathsf{let}\,\{x_i{=}e_i\}_{i=1}^n\,\mathsf{in}\,\lambda x.e$	$(n \geq 0)$	Val
	$\mathsf{let}\,\{x_i{=}e_i\}_{i=1}^n\,\mathsf{in}\,e$	$(e \in \mathsf{Val})$	Val
	$\mathsf{let}\,\{x_i{=}e_i\}_{i=1}^n\,\mathsf{in}\,e$	$(e \notin \mathsf{Val})$	Let

where the rightmost column presents the syntactic category at each row. e, e', e_1, ... will range over expressions (Exp). v, v', v_1, ... will range over values (Val). The name Srp ("sharp") denotes strict function application. (More on this later.) We will use the names of syntactic categories as if they were predicates that indicate membership. For example, $\mathsf{Val}(e)$ means $e \in \mathsf{Val}$.

We will drop the let-bindings in $\mathsf{let}\,\{x_i{=}e_i\}_{i=1}^n\,\mathsf{in}\,\lambda x.e$ when $n = 0$ and simply write $\lambda x.e$. Adding extra let-bindings to a value again results in a value. We inherit Launchbury's [16] standard restriction in that functions can only be applied to variables. As stated in [16], this does not reduce expressiveness because we also have let. The same argument holds for **strict application**, i.e., $e\#x$.

We quotient syntax up to α-equivalence of λ- and let-bound variables. For example, $\lambda x.x = \lambda y.y$. This design dismisses the need for local freshness check in [22].

Call $\mathsf{let}\,\{x_i{=}e_i\}_{i=1}^n\,\mathsf{in}\,\lambda x.e$ let**-surrounded abstractions**. In considering let-surrounded abstractions values, we follow Ariola and Felleisen [1]. Hereafter, unless stated otherwise, for $\mathsf{let}\,\{x_i{=}e_i\}_{i=1}^n\,\mathsf{in}\,\lambda x.e$ we assume that $n \geq 1$.

Definition 2: Define $fv(e)$ the **free variables** of e by:

$$fv(x) = \{x\} \qquad fv(\lambda x.e) = fv(e) \setminus \{x\}$$
$$fv(e\,x) = fv(e) \cup \{x\} \qquad fv(e\#x) = fv(e) \cup \{x\}$$
$$fv(\mathsf{let}\,\{x_i{=}e_i\}_{i=1}^n\,\mathsf{in}\,e) = (fv(e) \cup \textstyle\bigcup_{i=1}^n fv(e_i)) \setminus \{x_i\}_{i=1}^n.$$

Remark 1: We assume that all bound variables are distinct. As a result, we do not need to distinguish between $\mathsf{let}\,\{y_1 = e_1, y_2 = e_2\}\,\mathsf{in}\,\lambda x.e$ and $\mathsf{let}\,\{y_1 = e_1\}\,\mathsf{in}\,(\mathsf{let}\,\{y_2 = e_2\}\,\mathsf{in}\,\lambda x.e)$. We choose the former expression as a syntactic shorthand for the latter. We handle mutual recursion like Launchbury ([16, §3.2.1]). The implication is that y_1 and y_2 here can well be mutually recursive.

Notation 1: Write $e[y/x]$ for the usual capture-avoiding substitution of x by y in e. For example, $(\lambda y.x)[y/x] = \lambda y'.y$, when x, y, and y' are distinct variables. Moreover, write $(v)^{-\lambda}[y/\_]$ for $(\mathsf{let}\,\{x_i{=}e_i\}_{i=1}^n\,\mathsf{in}\,e)[y/x]$ where the value v is $\mathsf{let}\,\{x_i{=}e_i\}_{i=1}^n\,\mathsf{in}\,\lambda x.e$. In words, by $(v)^{-\lambda}[y/\_]$, we denote the result of "removing v's outmost λ and replacing all the occurrences of the argument bound to that λ – i.e., x in this case – by the variable y."

Definition 3: For a partial function f, define $dom(f) = \{x \mid f(x)\text{ defined}\}$. Call a partial function Γ mapping variable symbols to expressions and such that $dom(\Gamma)$ is finite, a **heap**. Γ, Δ, Θ, and Ξ will range over heaps. When $x \notin dom(\Gamma)$, define $(\Gamma, x \mapsto e)$ such that

- $(\Gamma, x \mapsto e)(x) = e$, and
- $(\Gamma, x \mapsto e)(y) = \Gamma(y)$ when $y \neq x$.

Besides,

- $(\Gamma, x_i \mapsto e_i)_{i=1}^1 = (\Gamma, x_1 \mapsto e_1)$, and
- $(\Gamma, x_i \mapsto e_i)_{i=1}^n = ((\Gamma, x_i \mapsto e_i)_{i=1}^{n-1}, x_n \mapsto e_n)$.

Definition 4: For $x \in dom(\Gamma)$, define $\Gamma[x \mapsto e]$ as:

- $\Gamma[x \mapsto e](x) = e$, and
- $\Gamma[x \mapsto e](y) = \Gamma(y)$ when $(y \neq x)$.

Furthermore,

- $\Gamma[x_i \mapsto e_i]_{i=1}^1 = \Gamma[x_1 \mapsto e_1]$, and
- $\Gamma[x_i \mapsto e_i]_{i=1}^n = (\Gamma[x_i \mapsto e_i]_{i=1}^{n-1})[x_n \mapsto e_n]$.

We overload the notation $\Gamma[x_i \mapsto e_i]_{i=1}^n$ for when $n = 0$ to be Γ itself.

Remark 2: Note the difference between the notations $\Gamma[x \mapsto e]$ and $(\Gamma, x \mapsto e)$: The former is well-formed when $x \in dom(\Gamma)$. To the contrary, the latter is well-formed when $x \notin dom(\Gamma)$. The former updates an existing binding whereas the latter augments the heap with a new one.

Definition 5: Let Γ_1 and Γ_2 be two heaps that bind the exact same variables, i.e., $dom(\Gamma_1) = dom(\Gamma_2)$. Define $\Gamma_1 \bowtie \Gamma_2$ as $\Gamma_1[x_i \mapsto v_i]_{i=1}^n$ where $\{x_i\}_{i=1}^n = \{x \mid \mathsf{Val}(\Gamma_2(x)) \wedge \neg\mathsf{Val}(\Gamma_1(x))\}$ and $v_i = \Gamma_2(x_i)$ for $1 \leq i \leq n$.

For the appropriate Γ_1 and Γ_2, the intuition is that $\Gamma_1 \bowtie \Gamma_2$ is Γ_1 itself some bindings of which are updated with values from Γ_2 that preserve semantics. This update is particularly useful when, over the same derivation, Γ_2 is done with the evaluation of certain variables that are still bound to non-value expressions in Γ_1. Note that not for every Γ_2 will $\Gamma_1 \bowtie \Gamma_2$ have the same semantics as Γ_1. See Example 1 for more.

Definition 6: Define the DLE big-step **operational semantics** $\Gamma : e \Downarrow \Delta : v$ as shown in Fig. 1 where:

- In $(\mathbf{var_x})$, by Definition 3, we assume $x \notin dom(\Gamma)$.
- In (**let**), 'x_i fresh' means $x_i \notin dom(\Gamma)$ and $x_i \notin fv(\Gamma(x))$ for $x \in dom(\Gamma)$, and similarly for Δ.[2]

Intuitively, $\Gamma : e \Downarrow \Delta : v$ is: "Trying to evaluate expression e in heap Γ will result in value v whilst the (probably manipulated) bindings are stored in Δ".

Remark 3: In line with Remark 1, for any expression $e = \mathsf{let}\,\{x_i{=}e_i\}_{i=1}^n\,\mathsf{in}\,e'$ to be evaluated, all occurrences of x_i's in e refer to the local let-bound x_i's. Especially, so are the free occurrences of x_i's, which are therefore needed to be distinguished from free variables of e. A particularly interesting impact this has in our system is how we define free variables for let expressions (Definition 2).

Notation 2: A **derivation** is the labelled tree — labelled with terms, heaps, and derivation rules from Definition 6 — that justifies a reduction $\Gamma : e \Downarrow \Delta : v$. $\Pi, \Pi_1, \Pi', \Pi_x, \ldots$ will range over derivations. Write '$\Gamma : e \Downarrow \Delta : v$' as shorthand for "$\Gamma : e \Downarrow \Delta : v$ is derivable". Write '$\Gamma : e \Downarrow_\Pi \Delta : v$' as shorthand for "$\Pi$ is a derivation of $\Gamma : e \Downarrow \Delta : v$". "_" will represent the wildcard in our notation, so that, by '$\Gamma : e \Downarrow \Delta : \_$', we are clarifying our lack of interest in the final value obtained after evaluating e in Γ. Write '$\Gamma : e \Downarrow_\Pi$' for '$\Gamma : e \Downarrow_\Pi \_ : \_$'. For a derivation $\Gamma : \_ \Downarrow \Delta : \_$, call Γ and Δ the **original** and **final** heaps, respectively.

[1]DLE = Distributed Lazy Evaluation

[2]Consistent with [16] and subsequent work, we allow the possibility that $x_i \in fv(e_j)$ or $x_i \in fv(e'_j)$ for $1 \leq i, j \leq n$.

$$\frac{}{\Gamma : v \Downarrow \Gamma : v}\ (\mathbf{val}) \qquad \frac{\Gamma : e \Downarrow \Delta : v}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto v) : v}\ (\mathbf{var_x}) \qquad \frac{\Gamma : e \Downarrow \Theta : v_e \quad \Theta : (v_e)^{-\lambda}[x/\_] \Downarrow \Delta : v}{\Gamma : e\,x \Downarrow \Delta : v}\ (\mathbf{app})$$

$$\frac{\Gamma : e \Downarrow \Theta_1 : v_e \quad \Gamma : x \Downarrow \Theta_2 : \_ \quad \Theta_1 \bowtie \Theta_2 : (v_e)^{-\lambda}[x/\_] \Downarrow \Delta : v}{\Gamma : e\#x \Downarrow \Delta : v}\ (\#)$$

$$\frac{(\Gamma, x_i \mapsto e_i)_{i=1}^n : e \Downarrow (\Delta, x_i \mapsto e'_i)_{i=1}^n : v \qquad \text{when } x_i \text{ fresh, } 1 \le i \le n}{\Gamma : \mathsf{let}\, \{x_i = e_i\}_{i=1}^n \,\mathsf{in}\, e \Downarrow \Delta : \mathsf{let}\, \{x_i = e'_i\}_{i=1}^n \,\mathsf{in}\, v}\ (\mathbf{let})$$

Fig. 1. Our Operational Semantics

All our rules but (#) are those of [6] except that we drop their (**seq**) rule for selective strictness. Note a particular consequence of our choice for let-surrounded abstractions to be values: In our (**let**) rule, we can put special care in place for 'garbage-collecting' the let-bindings which are of no relevance to heap Δ (i.e., those of x_i's).[3] The rule for let in [16], [24] does not garbage collect. For them, heaps are left with extra bindings after evaluation of let-expressions. The semantics in [16], [24] allows variables to escape their scope during evaluation, which is forbidden in our semantics. For example, in [16], evaluation of let $x = \lambda y.y$ in $\lambda z.(xz)$ finishes by adding $x \mapsto \lambda y.y$ to the original heap; this could then 'accidentally' bind x occurring elsewhere in the next expressions to evaluate. Launchbury is well aware of this issue and comments on it [16, §3.1]. To avoid 'accidental name-clash' in evaluations in his system, he imposes a pre-evaluation normalisation for expressions which renames all the variables already bound in heap. This normalisation is fine, if we just want to evaluate a single expression in a single heap. However, as discussed in detail in [5], [6], for reasoning about the evaluation of classes of programs and proving operational equivalences between them, a garbage-collecting rule for let is better.

Remark 4: Our rules are applicable based on a *best-fit* strategy: The final rule in $\Gamma : \mathsf{let}\,\{x_i{=}e_i\}_{i=1}^n\,\mathsf{in}\,v \Downarrow \Gamma : \mathsf{let}\,\{x_i{=}e_i\}_{i=1}^n\,\mathsf{in}\,v$ is always (**val**) as opposed to (**let**). In retrospect, in $\Gamma : \mathsf{let}\,\{x_i{=}e_i\}_{i=1}^n\,\mathsf{in}\,e \Downarrow \Delta : v$, the final rule is (**let**) when $\neg\mathsf{Val}(e)$. Here is an interesting related observation for expressions such as $e = \mathsf{let}\ x = (\mathsf{let}\ \{\}\ \mathsf{in}\ \lambda y.y)\ \mathsf{in}\ (\mathsf{let}\ \{\}\ \mathsf{in}\ \lambda t.t)$. Note first that, according to our convention explained in Remark 1, we identify e and $\mathsf{let}\ x = (\mathsf{let}\ \{\}\ \mathsf{in}\ \lambda y.y)\ \mathsf{in}\ \lambda t.t$. Therefore, the syntactic category of e is Val rather than Let. Clearly then (**val**) is the last rule applied in the evaluation of e. In other words, inclusion of (**val**) and (**let**) together in our operational semantics does not make it non-deterministic.

The intuition behind our (#) rule is that evaluation of e and x can happen independently and perhaps simultaneously. Hence, the implementation is free to assign each evaluation to a separate thread/processor. This freedom is the key to our task **distribution**. One reason why, out of the rules in Figure 1, we only present the mechanisation of (#) is this pivotal role of it. (This is done in Section V-A.) In words, (#) reads as follows: "Evaluate both e and x in the original heap; keep the first resulting value but discard the second; evaluate the suitably substituted expression in a heap which is suitably produced out of the resulting heaps of the two former steps; return both the resulting heap and expression intact." Refer back to Notation 1 for more on $(v_e)^{-\lambda}[x/\_]$.

Example 1: Let $\Gamma_1 = \{x_1 \mapsto \lambda t.t, x_2 \mapsto (\lambda x.x)\ x_1\}$ and $\Gamma_2 = \{x_1 \mapsto \lambda t.t, x_2 \mapsto \lambda x.\lambda y.x\}$. Then, by Definition 5, $\Gamma_3 = \Gamma_1 \bowtie \Gamma_2 = \{x_1 \mapsto \lambda t.t, x_2 \mapsto \lambda x.\lambda y.x\}$. Note, however, that Γ_1 and Γ_2 do not have the same semantics because $\Gamma_1 : x_2 \Downarrow \_ : \lambda t.t$ but $\Gamma_2 : x_2 \Downarrow \_ : \lambda x.\lambda y.x$. In other words, replacing x_2's binding over the process of building Γ_3 does not preserve the semantics of Γ_1. Corollary 4 describes a situation when $\bowtie$ does preserve semantics. □

III. Properties

This section provides half a dozen important properties of DLE. We start with Section III-A to discuss properties that, although fundamental, are still interesting on their own. We prove two observational equivalences using these properties. Then, in Section III-B, we present INMB along with the major results used to prove its validity for our system. INMB is the technique we use to prove Theorem 6.

A. Fundamental Properties and Equivalences

The most important result in this section is Theorem 1 that plays a key role in proving properties of DLE. Corollaries 1 and 2 demonstrate two immediate instances where the role of Theorem 1 is canonical.

Lemma 1: Suppose that $\Gamma : \_ \Downarrow \Delta : \_$. Then, $dom(\Gamma) = dom(\Delta)$. Besides, $\forall x \in dom(\Gamma).\ \mathsf{Val}(\Gamma(x)) \Rightarrow \mathsf{Val}(\Delta(x))$.

Proof: Rule-based induction. ■

Lemma 2: $\Theta_2(x) \in \mathsf{Val}$ implies $(\Theta_1 \bowtie \Theta_2)(x) \in \mathsf{Val}$.

Lemma 3 (Idempotence of $\bowtie$): $\Gamma \bowtie \Gamma = \Gamma$.

Theorem 1 (Determinism): Let $\Gamma : e \Downarrow \Delta_1 : v_1$ and $\Gamma : e \Downarrow \Delta_2 : v_2$. Then, $\Delta_1 = \Delta_2$ and $v_1 = v_2$.

Proof: Rule-based induction. ■

Definition 7 (Strict Equivalence [5], [6]): Define $e_1 \approxeq_s e_2$ by:

$$\forall \Gamma, v, \Delta.\ (\Gamma : e_1 \Downarrow \Delta : v) \Leftrightarrow (\Gamma : e_2 \Downarrow \Delta : v).$$

Intuitively, $e_1 \approxeq_s e_2$ when, given the same original heap, e_1 and e_2 always compute the same final value and final heap.

Corollary 1 (Strictness Dismissal): $x\#x \approxeq_s x\ x$.

[3]This particular use of the term 'garbage-collection' is due to Launchbury [16, §6.2].

Proof: Suppose that $\Gamma : x\ x \Downarrow \Delta : v$. The derivation takes the following form:

$$\frac{\Gamma : x \Downarrow_{\Pi_1} \Theta : v_x \quad \Theta : (v_x)^{-\lambda}[x/\_] \Downarrow_{\Pi_2} \Delta : v}{\Gamma : x\ x \Downarrow \Delta : v}\ (\mathbf{app}).$$

We use Π_1 and Π_2 to build a derivation for $\Gamma : x\#x \Downarrow \Delta : v$ as follows:

$$\frac{\Gamma : x \Downarrow_{\Pi_1} \Theta : v_x \quad \begin{array}{l}\Gamma : x \Downarrow_{\Pi_1} \Theta : \_ \\ \quad \Theta \bowtie \Theta : (v_x)^{-\lambda}[x/\_] \Downarrow_{\Pi_2} \Delta : v\end{array}}{\Gamma : x\#x \Downarrow \Delta : v}\ (\#).$$

Note that, according to Lemma 3, $\Theta \bowtie \Theta = \Theta$. Furthermore, due to Determinism (Theorem 1), if $\Gamma : x\#x \Downarrow \Delta' : v'$, then $\Delta = \Delta'$ and $v = v'$. The proof is similar for the other direction. The result follows by Definition 7. ∎

In words, Corollary 1 dismisses the need for spawning a new process for evaluating strict self-applications. This goes well with the intuition that, in self-application, evaluation of the function has already addressed the evaluation of the argument too.

Corollary 2: $(e\#x)\#x \approxeq_s (e\#x)\ x$.

Proof: Suppose that $\Gamma : (e\#x)\#x \Downarrow \Delta : v$. The derivation takes the form:

$$\frac{\Gamma : e\#x \Downarrow_{\Pi_1} \Theta' : v' \quad \begin{array}{l}\Gamma : x \Downarrow \Theta_2 : \_ \\ \quad \Theta' \bowtie \Theta_2 : (v')^{-\lambda}[x/\_] \Downarrow_{\Pi_2} \Delta : v\end{array}}{\Gamma : (e\#x)\#x \Downarrow \Delta : v}\ (\#)$$

where Π_1 is

$$\frac{\Gamma : e \Downarrow \Theta_1 : v_e \quad \begin{array}{l}\Gamma : x \Downarrow \Theta_2 : \_ \\ \quad \Theta_1 \bowtie \Theta_2 : (v_e)^{-\lambda}[x/\_] \Downarrow \Theta' : v'\end{array}}{\Gamma : e\#x \Downarrow \Theta' : v'}\ (\#).$$

Note now that

$$\begin{array}{ll}\forall y \in dom(\Gamma).\mathsf{Val}(\Theta_2(y)) \Rightarrow & \text{Lemma 2}\\ \qquad\qquad \mathsf{Val}((\Theta_1 \bowtie \Theta_2)(y)) \Rightarrow & \text{Lemma 1}\\ \qquad\qquad \mathsf{Val}(\Theta'(y)) \Rightarrow & \\ \nexists y.\ \mathsf{Val}(\Theta_2(y)) \wedge \neg\mathsf{Val}(\Theta'(y)) \Rightarrow \Theta' \bowtie \Theta_2 = \Theta'. & \end{array}$$

We build $\Gamma : (e\#x)\ x \Downarrow \Delta : v$ using Π_1 and Π_2:

$$\frac{\Gamma : e\#x \Downarrow_{\Pi_1} \Theta' : v' \quad \Theta' : (v')^{-\lambda}[x/\_] \Downarrow_{\Pi_2} \Delta : v}{\Gamma : (e\#x)\ x \Downarrow \Delta : v}\ (\mathbf{app}).$$

The result follows by Determinism (Theorem 1). ∎

Again, Corollary 2 dismisses the need for spawning a new process for the task that is already addressed. The important observation is that we easily prove such intuitive equivalences because of the simplicity in our semantics. In particular, in our proofs, we are not burdened with the bookkeeping information for interprocess communication. This would have not been the case if we wanted to try the same proofs under the system of Sánchez-Gil et al. [21]. See the closing paragraph of Section VI-A for a related discussion.

B. *The Induction Principle*

In this section we present a formulation of INMB and give a summary of what it takes to prove it for the operational semantics of Definition 6. We begin by a necessary definition:

Definition 8: Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$. Define $\mathit{diff}(\Pi)$ by:

$$\mathit{diff}(\Pi) = \{x \in dom(\Gamma) \mid \Gamma(x) \neq \Delta(x)\}.$$

Intuitively, $\mathit{diff}(\Pi)$ is those variables bindings of which are manipulated over Π. We now get to a formulation of INMB

$$\frac{\begin{array}{c}[P(\Pi); \mathit{diff}(\Pi) = \varnothing] \\ \wedge \\ [\forall k.(P(\Pi); \|\mathit{diff}(\Pi)\| = k) \Rightarrow \\ \qquad (P(\Pi); \|\mathit{diff}(\Pi)\| = k+1)]\end{array}}{\forall n.\ P(\Pi); \|\mathit{diff}(\Pi)\| = n}\ (\textsc{Inmb}).$$

In words, (INMB) states that: "$P(\Pi)$ holds if: (1) $P(\Pi)$ holds when no binding gets manipulated over derivation Π. And, (2) validity of $P(\Pi)$ when there are k manipulated bindings in derivation Π implies validity of $P(\Pi)$ when there are $k+1$ manipulated bindings."

INMB has a deceivingly simple statement. The proof of its validity is, however, by no means trivial. The rest of this section retraces the major steps.

Theorem 2: INMB is valid for the operational semantics in Definition 6.

Proof: Using Theorems 3, 4 and 5. ∎

In order to state the theorems used in the proof of Theorem 2, we need to lay down some pieces of terminology.

Definition 9: Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$. Write $V(\Pi)$ for the set of $x \in dom(\Gamma)$ such that Π contains an instance of $(\mathbf{var_x})$.

Definition 10: Call x **atomic** in Γ when there exist Δ_x, v_x, and Π_x such that $\Gamma : x \Downarrow_{\Pi_x} \Delta_x : v_x$ and $\mathit{diff}(\Pi_x) = \{x\}$. Write $\mathit{atomic}(\Gamma)$ for the set of atomic variable symbols in Γ.

The intuition is that a variable is atomic when "its evaluation only manipulates the binding of the variable itself." We next extend Definition 4 to full derivations:

Definition 11: Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$ and suppose Π contains no instance of $(\mathbf{var_x})$. Then, for any e', define $\Pi[x \mapsto e']$ to be the labelled tree obtained from Π by replacing every heap Θ appearing in Π with $\Theta[x \mapsto e']$ if $x \in dom(\Theta)$; otherwise, we leave Θ unchanged.

We will prefer the following slightly more succinct characterisation of atomicity:

Lemma 4: $x \in \mathit{atomic}(\Gamma)$ if and only if there exist v_x and Π_x such that $\Gamma : x \Downarrow_{\Pi_x} \Gamma[x \mapsto v_x] : v_x$ and $\mathit{diff}(\Pi_x) = \{x\}$.

Here is our last ingredient for Theorems 3, 4 and 5:

Definition 12: Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$ such that $x \in V(\Pi)$. Suppose also that $x \in \mathit{atomic}(\Gamma)$, so that in particular $\Gamma : x \Downarrow_{\Pi_x} \Delta_x : v_x$ for some Π_x, Δ_x, and v_x.

We define a labelled tree $\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle$ by inductively transforming Π based on its final rule. For each subcase,

- when $x \notin \mathit{dom}(\Gamma)$ or $x \in V(\Pi) \setminus \mathit{atomic}(\Gamma)$, define
$$\Pi\llangle x \mapsto v_x \rrangle = \Pi. \qquad (1)$$
- when $x \notin V(\Pi)$, define
$$\Pi\llangle x \mapsto v_x \rrangle = \Pi[x \mapsto v_x]. \qquad (2)$$

The subcases where $x \in \mathit{atomic}(\Gamma) \cap V(\Pi)$ are defined below.

- $(\mathbf{var_x})$. Π takes the form
$$\dfrac{\begin{array}{c}\vdots\\ \Gamma' : e_x \Downarrow \Gamma' : v_x\end{array}}{(\Gamma', x \mapsto e_x) : x \Downarrow (\Gamma', x \mapsto v_x) : v_x}\ (\mathbf{var_x}).$$
Define $\Pi\llangle x \mapsto v_x \rrangle$ to be
$$\dfrac{\dfrac{}{\Gamma' : v_x \Downarrow \Gamma' : v_x}\ (\mathbf{val})}{(\Gamma', x \mapsto v_x) : x \Downarrow (\Gamma', x \mapsto v_x) : v_x}\ (\mathbf{var_x}).$$
- $(\mathbf{let})$ and $(\mathbf{var_y})$ for y other than x. Π takes the form
$$\dfrac{\begin{array}{c}\vdots\ \Pi'\\ \Gamma' : e' \Downarrow \Delta' : v'\end{array}}{\Gamma : e \Downarrow \Delta : v}\ (\mathbf{r})$$
where $(\mathbf{r}) \in \{(\mathbf{var_y}), (\mathbf{let})\}$. Define $\Pi\llangle x \mapsto v_x \rrangle$ to be
$$\dfrac{\begin{array}{c}\vdots\ \Pi'\llangle x \mapsto v_x \rrangle\\ \Gamma'[x \mapsto v_x] : e' \Downarrow \Delta'[x \mapsto v_x] : v'\end{array}}{\Gamma[x \mapsto v_x] : e \Downarrow \Delta[x \mapsto v_x] : v}\ (\mathbf{r}).$$

The transformation for $(\mathbf{app})$ and $(\#)$ takes the same bottom-up fashion of the latter case. Note that $(\mathbf{val})$ can never be the final rule when $x \in \mathit{atomic}(\Gamma) \cap V(\Pi)$ because $V(\Pi) = \varnothing$ in this case. Hence, the recipe for this case is (2) where $\Pi\llangle x \mapsto v_x \rrangle = \Pi[x \mapsto v_x] = \Pi$ especially. (See Definition 4.) □

Theorems 3 and 4, respectively, explain how to *remove* and *restore* an atomic variable from a derivation to obtain another (valid) derivation.

Theorem 3: Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$ and $x \in V(\Pi)$. Suppose $x \in \mathit{atomic}(\Gamma)$; so in particular $\Gamma : x \Downarrow_{\Pi_x} \Gamma[x \mapsto v_x] : v_x$ for some v_x. Then, $\Delta(x) = v_x$ and $\Gamma[x \mapsto v_x] : e \Downarrow_{\Pi\llangle x \mapsto v_x \rrangle} \Delta : v$. Furthermore,
$$\begin{array}{c}\mathit{diff}(\Pi\llangle x \mapsto v_x \rrangle) = \mathit{diff}(\Pi) \setminus \{x\}\\ V(\Pi\llangle x \mapsto v_x \rrangle) \cup V(\Pi_x) = V(\Pi)\\ x \in V(\Pi\llangle x \mapsto v_x \rrangle).\end{array}$$

Theorem 4: Suppose $x \in \mathit{atomic}(\Gamma)$; so, in particular, $\Gamma : x \Downarrow_{\Pi_x} \Gamma[x \mapsto v_x] : v_x$ for some v_x, and Π_x such that $\mathit{diff}(\Pi_x) = \{x\}$. Suppose also that $\Gamma\llangle x \mapsto v_x \rrangle : e \Downarrow_{\Pi'} \Delta : v$ and $x \in V(\Pi')$. Then, $\Gamma : e \Downarrow_\Pi \Delta : v$ for a Π such that $\Pi\llangle x \mapsto v_x \rrangle = \Pi'$ and $x \in V(\Pi)$.

Theorem 5 proves that any derivation either manipulates no bindings, or, of the manipulated bindings at least one binds an atomic variable in the original heap. (See Notation 2 for the definition of original heap.)

Theorem 5: Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$. Then exactly one of the following possibilities holds: $\mathit{diff}(\Pi) = \varnothing$; or there exists an $x \in \mathit{diff}(\Pi)$ such that $x \in \mathit{atomic}(\Gamma)$.

IV. Heap Bisimilarity

The results in this section give criteria for when using one heap instead of the other does not risk correctness. Bisimilar heaps can be passed around in place of each other because they are guaranteed to always produce the same results. We begin by introducing our transition system. Then, we define a relation on the set of heaps that we later show is a bisimulation for our transition system:

Definition 13: Let $\mathcal{H}$ be the set of all heaps. Define $\xrightarrow{(e,v)}$ for the transition system $\mathscr{T}_{\mathcal{H}} = (\mathcal{H}, \mathsf{Exp} \times \mathsf{Val}, \xrightarrow{(.,.)})$ such that
$$\Gamma_1 \xrightarrow{(e,v)} \Gamma_2 \text{ when } \Gamma_1 : e \Downarrow \Gamma_2 : v.$$

Definition 14: Call Γ_1 and Γ_2 **analogous** ($\Gamma_1 \approx \Gamma_2$) when:
$$\forall e, v.\ \Big((\Gamma_1 : e \Downarrow \_ : v) \Leftrightarrow (\Gamma_2 : e \Downarrow \_ : v)\Big).$$

In words, two heaps are called analogous when they "always evaluate the same expression to the same value." Note that this definition is indeed also taking irreducibility into consideration: If the evaluation of an expression e (such as let $z = \lambda x.(xx)$ in zz) in one of the two analogous heaps does not terminate, e's evaluation should not terminate in the other heap either. Likewise, if an expression's evaluation halts abruptly in one heap – say, due to unavailability of a requested binding – the same evaluation should halt abruptly for the other analogous heap too.

Lemma 5: $\approx$ is an equivalence relation (reflexive, transitive, symmetric).

Theorem 6: Suppose that $\Gamma : \_ \Downarrow \Delta : \_$. Then, $\Gamma \approx \Delta$.

Proof: Using INMB. ■

Corollary 3: Let $\Gamma : \_ \Downarrow \Delta_1 : \_$ and $\Gamma : \_ \Downarrow \Delta_2 : \_$. Then, $\Delta_1 \approx \Delta_2$.

Proof: Using Theorem 6 and Lemma 5. ■

Corollary 4 (Symmetry of $\bowtie$): When $\Gamma : e_1 \Downarrow \Delta_1 : \_$ and $\Gamma : e_2 \Downarrow \Delta_2 : \_$ for some arbitrary e_1 and e_2, $\Delta_1 \bowtie \Delta_2 = \Delta_2 \bowtie \Delta_1$.

Proof: Note first that, by Lemma 1, $\mathit{dom}(\Delta_1) = \mathit{dom}(\Gamma) = \mathit{dom}(\Delta_2)$ and both $\Delta_1 \bowtie \Delta_2$ and $\Delta_2 \bowtie \Delta_1$ are well-formed. Suppose that $\Gamma : e_1 \Downarrow_{\Pi_1} \Delta_1 : \_$ and $\Gamma : e_2 \Downarrow_{\Pi_2} \Delta_2 : \_$. We distinguish between four cases:

- $x \notin \mathit{diff}(\Pi_1) \cup \mathit{diff}(\Pi_2)$ Then, $(\Delta_1 \bowtie \Delta_2)(x) = \Gamma(x) = (\Delta_2 \bowtie \Delta_1)(x)$.
- $x \in \mathit{diff}(\Pi_1) \setminus \mathit{diff}(\Pi_2)$ Then, $(\Delta_1 \bowtie \Delta_2)(x) = \Delta_1(x) = (\Delta_2 \bowtie \Delta_1)(x)$.
- $x \in \mathit{diff}(\Pi_2) \setminus \mathit{diff}(\Pi_1)$ Then, $(\Delta_2 \bowtie \Delta_1)(x) = \Delta_2(x) = (\Delta_1 \bowtie \Delta_2)(x)$.
- $x \in \mathit{diff}(\Pi_1) \cap \mathit{diff}(\Pi_2)$ Let $\Delta_1(x) = v_{1x}$ and $\Delta_2(x) = v_{2x}$. By Corollary 3, $\Delta_1 \approx \Delta_2$. That is, by Definition 14, $v_{1x} = v_{2x}$.

The result follows. ■

Corollary 4 legislates changing the bowtie ($\bowtie$) order of one heap into another. One might want to do this for various reasons, including: proximity of the bindings that are subject to update upon the bowtie, their locality issues, and the channelling costs involved.

Lemma 6: $\approx$ is a bisimulation for $\mathscr{T}_{\mathcal{H}}$.

Proof: Fix Γ_1 and Γ_2 s.t. $\Gamma_1 \approx \Gamma_2$ and $\Gamma_1 \xrightarrow{(e,v)} \Delta_1$ for some heap Δ_1. If we show that there exists a heap Δ_2 such that $\Gamma_2 \xrightarrow{(e,v)} \Delta_2$ and $\Delta_1 \approx \Delta_2$, the result follows by symmetry.

By construction (Definition 13), $\Gamma_1 \xrightarrow{(e,v)} \Delta_1$ means $\Gamma_1 : e \Downarrow_{\Pi_1} \Delta_1 : v$ for some Π_1. Given that $\Gamma_1 \approx \Gamma_2$, by construction (Definition 14), it follows that there exists a heap Δ_2 such that $\Gamma_2 : e \Downarrow \Delta_2 : v$. By Theorem 6, $\Gamma_1 \approx \Delta_1$ and $\Gamma_2 \approx \Delta_2$. The result follows by transitivity of $\approx$ (Lemma 5). ■

Theorem 7: $\approx$ is the bisimilarity for $\mathscr{T}_{\mathcal{H}}$.

Proof: Using Lemma 6 and Definition 13. ■

Due to their number of bindings, size of expressions they bind, and many other reasons, passing heaps around can become expensive – especially, when remote inter-communication is entailed. A special occasion where that might be needed is when updating a local heap with bindings of another. Theorem 7 dismisses that need when the heaps involved are known to be analogous.

V. Mechanisation

This section first gives an overview about the mechanisation approach that we used for this paper. We next report the benefits we gained from the mechanisation exercise. At last, we present our new mechanisation component that we developed to use together with the existing ones that we reused.

Figure 2 is a snapshot of a Scala session that shows the successful reduction of $\Gamma : (\lambda z_1.\lambda z_2.z_1)\#x_1$, where Γ (denoted in the code by `G`) is $\{x_1 \mapsto \lambda x.x, x_2 \mapsto x_1\ y\}$. Generally, in this framework, `g <::> e` outputs the derivation $\Gamma : e \Downarrow$, when derivable; it produces an error message otherwise. For example, attempting to evaluate x_3 in the above Γ will throw the following exception:

```
Evaluating unbound variable x3 in heap G
```

In our former work [8] we promoted **component-based** mechanisation (CBM) of PLs. CBM is the familiar Component-Based Software Engineering (CBSE) applied in the field of PL mechanisation. The idea leads to a separation of concerns between the PL developers, mechanisation component vendors, and Language Definitional Frameworks (LDFs): An LDF provides the necessary environment and **standards** for the development of the components as well as PL implementation using them[4]; the component vendors enjoy the LDF environment to deliver components; and, the PL implementers mix and match the available PL components to get their desired PL mechanised. Such components need to enjoy a *plug-and-play* nature: That is, the PL implementer simply picks their desirable PL components from an available repository; and, as soon as they (correctly) plug the chosen components together, they are ready to kick-off the experimentation with their PL.

[4]See [23, Chapter 17] and [20, Chapter 10] for a discussion on the role the development environment and standards in CBSE.

In the absence of the right component, it suffices to only develop the missing component and reuse the available ones. For example, to get the DLE mechanised in a CBM fashion, one plugs together components which correspond to: `Var` and $(\mathbf{var_x})$, `App` and $(\mathbf{app})$, `Srp` and $(\#)$, `Val` and $(\mathbf{val})$, as well as `Let` and $(\mathbf{let})$. Out of those components, however, we had to develop the `Srp` one whilst reusing the other four. In Section V-A, we take a deeper look into how our mechanisation of the $(\#)$ rule manages running its two left-most premisses in-parallel. Plugging components together usually involves preparing some *glue code* as well. Details of how to develop the glue code in this case are discussed in [8].

The benefit of using mechanisation goes well beyond sole PL implementation. For example, in our initial theory-work, we had originally decided for the rightmost assumption of our $(\#)$ to be $\Theta_1 \bowtie \Theta_2 : (v_e)^{-\lambda}[\mathbf{v_x}/\_]$ (where $\Gamma : x \Downarrow \_ : v_x$). It turns out that, we did not notice the mistake until our mechanisation revealed it that $(v_e)^{-\lambda}[v_x/\_]$ produces syntactically invalid expressions when $v_e = \lambda y.(e'\ y)$ for some expression e' – thanks to Scala's static type checking.

Mechanisation is also suitable when it is used in tandem with the theoretical development. For instance, our mechanisation made us understand the necessity of the $dom(\Gamma_1) = dom(\Gamma_2)$ condition in Definition 5 far before our theory comes to that need. Our understanding is that mechanisation helps the PL design avoid mistakes that are hard to envision but easier to spot in action. The plus point of our CBM technology is that it makes rapid PL implementation very feasible. As such, it can indeed be a lightweight assistant to PL development. See Klein et al. [14] for a comprehensive discussion.

A. The Executable $(\#)$ Rule

A winning feature of CBSE (and consequently CBM) is the **independence** nature of components. The benefits of this nature for software development are already well explored in the literature. (See [23], [20].) The same nature enables us here to unveil our mechanisation for the $(\#)$ of Definition 6 – i.e., `HBSharpRuleVal` below – without discussing the other components. Our follow-up explanation aims to make the tight correspondence between `HBSharpRuleVal` and $(\#)$ clear.

```
scala> println(g <::> (\[DLEExp]("z1", "z2")("z1") <#> "x1"))
                                       ------------------------------------------- (val)
                                       {x2 |-> x1 y}: \x.x ==> {x2 |-> x1 y}: \x.x
------------------------------- (val)  --------------------------------------------------- (var)  ------------------------------------------------------------------------- (val)
G: \z1.\z2.z1 ==> G: \z1.\z2.z1        G: x1 ==> {x2 |-> x1 y, x1 |-> \x.x}: \x.x                 {x2 |-> x1 y, x1 |-> \x.x}: \z2.x1 ==> {x2 |-> x1 y, x1 |-> \x.x}: \z2.x1
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------(#)
G: (\z1.\z2.z1) # x1 ==> {x2 |-> x1 y, x1 |-> \x.x}: \z2.x1
```

Fig. 2. Mechanised DLE in Action

```
1  object HBSharpRuleVal {
2    def apply[
3      Exp <: LazyExp[Exp],
4      Val <: IVal[Exp] with Exp,
5      Let <: ILet[Exp] with Exp,
6      Srp <: ISrp[Exp] with Exp,
7      OS  <: OpSem[Exp]{
8        type Conf = HBConf[Exp]
9        type Node = HBNode[Exp]}
10   ](g: Heap[Exp], ex: Srp) = {
11     val (e, x) = (ex.e, ex.x)
12
13     val pi1f = Future {g <::> e}
14     val pi2f = Future {g <::> x}
15
16     val node = for {
17       pi1 <- pi1f
18       pi2 <- pi2f
19
20       (t1, ve, t2) = (pi1.g2, pi1.e2, pi2.g2)
21
22       pi3 <- Future {(t1 |><| t2) <::> (ve ^-\ ex.x)}
23       (d, v) = (pi3.g2, pi3.e2)
24     } yield new HBSrpNode[Exp](pi1, pi2, pi3,
25                                g, ex, d, v)
26
27     Await.result(node, Inf)
28   }
29 }
```

Lines 3 to 9 manifest the usage **protocol** of `HBSharpRuleVal`. That is, they specify the wide range of syntax and semantics that the component can be used with. Lines 3 to 6 state that it can work with any syntax mechanisation for laziness that provides the following three type constructors – expressed through the respective upper bound conditions: `Val`, `Let`, and `Srp`. Similarly, lines 7 to 9 place restrictions on the operational semantics that this rule can be a part of. In short, they state that both the semantics and the output derivation trees (like the one in Figure 2) need to be heap-based. That is what the prefix `HB` of `HBSharpRuleVal` indicates.

The arguments in line 10 are the heap (`g` for Γ) and the expression (`ex` for $e\#x$) this rule is responsible for. Line 11 extracts the two parts of a strict function application: the function (`e`) and the argument to which the function is applied (`x`). Lines 13 and 14 inform Scala that the evaluations $\Gamma : e$ and $\Gamma : x$ are likely to be examined asynchronous to the rest of the computation.

The real asynchronous computation takes place in the `for`-comprehension of lines 16 to 25 where `pi1` and `pi2` are run in parallel. (Scala's `for`-comprehensions are monadic entities like the `do` notation of HASKELL.) Note that `pi3` – which represents the rightmost branch of the $(\#)$ – depends on pieces of `pi1` and `pi2`. Hence, Scala needs to wait for both the latter (parallel) evaluations before it can start that of `pi3`. Lines 24 and 25 specify how to build the right derivation out of the pieces gathered over the `for`-comprehension. Finally, line 27 instructs Scala to wait until the `for`-comprehension is done and return the derivation tree specified in lines 24 and 25.

VI. Related Work

A. Semantics

Launchbury [16] was the first to provide an operational semantics for lazy evaluation. Nakata and Hasegawa [18] add variable hygiene in the style of Sestoft [22] to his work. Our garbage-collecting (let) rule leaves no need for such a hygiene for us. Hall et al. [9], [2] build on Launchbury's work to get a small-step operational semantics for parallel lazy evaluation in the presence of selective strictness. Hidalgo-Herrero and Ortega-Mallén [11] do the same but for distributed lazy evaluation. This latter work was then followed by Sánchez-Gil et al. [21] to provide the first big-step operational semantics for the same setting. Our DLE enjoys remarkable simplicity in comparison to Sánchez-Gil et al. The other related work is that by van Eekelen and de Mol [24], which was the first to study selective strictness in a big-step semantics. All the above works (except that of ours) suffer from increase of heap expressiveness upon evaluation of let-expressions (IHEELE). This was first noticed and repaired in [5] but with restrictions on the language expressiveness. These restrictions were removed in [6] to form the first big-step semantics for lazy evaluation (in the presence of selective strictness) that does not suffer from IHEELE. Our present work takes the same route to provide a simple big-step semantics for distributed lazy evaluation.

There is a subtle difference between our system and that of Sánchez-Gil et al. [21] that is worth more elaboration: Our strict application is similar to their *parallel application* in that, upon encountering an $e\#x$, we both eagerly spawn two derivations – one for evaluating e and another for x. However, in parallel application, the side effects of the latter evaluation are discarded unless x is indeed requested over the former evaluation. In strict application, to the contrary, we always perform the $\Theta_1 \bowtie \Theta_2$, regardless of whether x was needed over $\Gamma : e \Downarrow \Theta_1 : \_$ or not. In other words, one can easily verify it that $e\#x$ in our system affects the heap similar to x seq $(e\ x)$ under traditional selective strictness. That is, they both cause the evaluation of the exact same variables. Interestingly enough, however, our $(\#)$ manages coordination in a way that suits task distribution. Whilst that is not possible under say (**let!**) of [24] (for selective strictness).

We would like here to elaborate on our comment about our work being considerably simpler than its sole predecessor: Figure 3 illustrates the rules of Sánchez-Gil et al. [21] that handle their parallel application.[5] The fact that our single $(\#)$

[5]In fact, they also have four other rules for λ-abstractions, variables, function applications, and let-expressions. Whilst, for these four rules too Sánchez-Gil et al. add complications to those of Launchbury, we only present their parallel application rules here. This is because their other four rules are not remarkably more complicated than their Launchbury counterparts.

(Process creation) $\Gamma' = \Gamma \cup \Gamma^B \cup \{\theta \mapsto x\#y\}$, if $\neg\mathsf{d}(x, \Gamma')$

$$\frac{\overline{\Gamma} + \{i \mapsto y, o \mapsto \eta(x)\ z, z \mapsto i\} + \eta(\mathsf{nh}(x, \Gamma')) : \theta \mapsto o \ \Downarrow\ \overline{\Delta} : \theta \mapsto W}{\overline{\Gamma} : \theta \mapsto x\#y \ \Downarrow\ \overline{\Delta} : \theta \mapsto W}$$

(Proc. creat. dem.) $\Gamma' = \Gamma \cup \Gamma^B \cup \{z \mapsto E', \theta \mapsto x\#y\}$, if $\mathsf{d}(x, \Gamma') \ \wedge\ \mathsf{fd}(x, \Gamma') = z \mapsto E' \ \wedge z \neq \theta$

$$\frac{\overline{\Gamma} + \{\theta \overset{B(z)}{\mapsto} x\#y\} : z \mapsto E' \ \Downarrow\ \overline{\Delta} + \{\theta \overset{B(z)}{\mapsto} x\#y\} : z \mapsto W' \qquad \overline{\Delta} + \{z \mapsto W'\} : \theta \mapsto x\#y \ \Downarrow\ \overline{\Theta} : \theta \mapsto W}{\overline{\Gamma} + \{z \mapsto E'\} : \theta \mapsto x\#y \ \Downarrow\ \overline{\Theta} : \theta \mapsto W}$$

(Communication) if $\neg\mathsf{d}(W, \overline{\Delta})$

$$\frac{\overline{\Gamma} + \{\theta \overset{B(ch)}{\mapsto} ch\} : ch \mapsto E \ \Downarrow\ \overline{\Delta} + \{\theta \overset{B(ch)}{\mapsto} ch\} : ch \mapsto W}{\overline{\Gamma} + \{ch \mapsto E\} : \theta \mapsto ch \ \Downarrow\ \mathsf{sat}(\overline{\Delta} \cup \eta(\mathsf{nh}(W, \overline{\Delta})) : \theta \mapsto \eta(W))}$$

(Comm. demand) $\Delta' = \Delta \cup \Delta^B \cup \{z \mapsto E'\}$, if $\mathsf{d}(W, \Delta') \ \wedge\ \mathsf{fd}(W, \Delta') = z \mapsto E'$

$$\frac{\begin{array}{c}\overline{\Gamma} + \{\theta \overset{B(ch)}{\mapsto} ch\} : ch \mapsto E \Downarrow \overline{\Delta} + \{z \mapsto E', \theta \overset{B(ch)}{\mapsto} ch\} : ch \mapsto W \\ \overline{\Delta} + \{ch \mapsto W, \theta \overset{B(z)}{\mapsto} ch\} : z \mapsto E' \Downarrow \overline{\Theta} + \{ch \mapsto W, \theta \overset{B(z)}{\mapsto} ch\} : z \mapsto W' \\ \overline{\Theta} + \{z \mapsto W', ch \mapsto W\} : \theta \mapsto ch \Downarrow \overline{\Lambda} : \theta \mapsto W''\end{array}}{\overline{\Gamma} + \{ch \mapsto E\} : \theta \mapsto ch \Downarrow \overline{\Lambda} : \theta \mapsto W''}$$

where $z \in \mathit{Var}$ and $o, i, \in \mathit{Chan}$ are fresh names, and η is a fresh renaming

Fig. 3. Process Creation and Communication Rules of Sánchez-Gil et al. [21]

is much simpler than this collection of rules is clear. Most of the simplicity in our (#) comes from the fact that its two left-most premisses independently (perhaps in par). Whereas in Figure 3 four rules are employed for creation of processes and communication between them: The rules **(Process creation)** and **(Proc. creat. dem.)** deal with process creation when that is "feasible" and when "dependencies" are detected, respectively. Likewise, the rules **(Communication)** and **(Comm. demand)** handle process communication in the absence and presence of "dependencies," respectively. (See [21] for their definitions of dependency and feasibility.)

B. *Mechanisation*

CBM is available under PLanCompS [13], Polyglot [19], GLoo [17], and SugarHaskell [4]. PLanCompS assumes a unique semantics for each of its syntax components. The other three works of this group target mechanisation through syntactic desugaring into a core PL. MontiCore [15] and Neverlang 2 [3] are LDFs for DSL mechanisation. Although not designed for this particular purpose, the "compositional visitors" of MontiCore can be used for CBM. Finally, the focus in Neverlang is on component-based compiler generation for DSLs where there is no emphasise on a formal semantics.

VII. Conclusion and Future Work

In this paper, we present a new operational semantics for distributed lazy evaluation that is remarkably simpler than its sole predecessor. It also enjoys several interesting properties such as validity of INMB, heap bisimilarity, and a notion of determinism (despite its distributed nature). The simplicity in our system is not confined to the statement of our semantics rules. We showcase the simplicity of proving observational equivalences with two intuitive results. Moreover, we explain the mechanisation of our system. Finally, we discuss how our mutual development of mathematics and mechanisation helped us address some unforeseen issues.

Whilst our (#) indeed models distribution, it is less than ideal in comparison to parallel application. (See Section VI.) A future work would then be an operational semantics with enough simplicity that can model parallel application and the study of its properties. That would entail an on-the-fly mechanism for checking looked up variables. We anticipate that techniques such as the "busy signal" trick of Hoffmann and Shao [12] would form a good starting point. Another future work can be exploring further observational equivalences under our current system. Several threads for future work on the mechanisation part are also possible: A crucial one can be enriching the PL with new constructs and updating the mechanisation for empirical studies on the benefits or losses of our flavour of task distribution. Another could be mechanisation of the intermediate results we obtained over our theoretical development reported in this paper. Finally, trying the mechanisation of this work under other CBM frameworks is yet another subject for later research.

References

[1] Z. Ariola and M. Felleisen, *The Call-by-Need λ-Calculus*, J. Func. Prog. **7** (1997), no. 3, 265–301.

[2] C. Baker-Finch, D. King, and P. Trinder, *An Operational Semantics for Parallel Lazy Evaluation*, Proc. 5th ACM SIGPLAN Int. Conf. Func. Prog., September 2000, pp. 162–173.

[3] W. Cazzola and E. Vacchi, *Neverlang 2 — Componentised Language Development for the JVM*, Proc. 12th Int. Conf. Soft. Composition (W. Binder, E. Bodden, and W. Löwe, eds.), LNCS, vol. 8088, Springer, June 2013, pp. 17–32.

[4] S. Erdweg, F. Rieger, T. Rendel, and K. Ostermann, *Layout-sensitive Language Extensibility with SugarHaskell*, Proc. 5th ACM SIGPLAN Symp. on Haskell (J. Voigtländer, ed.), ACM, September 2012, pp. 149–160.

[5] S. H. Haeri, *Reasoning about Selective Strictness: Operational Equivalence, Heaps and Call-by-Need Evaluation, New Inductive Principles*, Master's thesis, Mathematical and Computer Sciences, Heriot-Watt University, 2009.

[6] ———, *Observational Equivalence and a New Operational Semantics for Lazy Evaluation with Selective Strictness*, Proc. Int. Conf. Theo. & Math. Found. Comp. Sci. (TMFCS-10), 2010, pp. 143–150.

[7] ———, *A New Operational Semantics for Distributed Lazy Evaluation*, Technical Report TR2013-1, Inst. Soft. Sys., Hamburg U. Tech., April 2013, http://www.sts.tuhh.de/tech-reports/2013/haer2013.pdf.

[8] S. H. Haeri and S. Schupp, *Reusable Components for Lightweight Mechanisation of Programming Languages*, Proc. 12th Int. Conf. Soft. Composition (W. Binder, E. Bodden, and W. Löwe, eds.), LNCS, vol. 8088, Springer, June 2013, pp. 1–16.

[9] J. Hall, C. Baker-Finch, P. Trinder, and D. King, *Towards an Operational Semantics for a Parallel Non-Strict Functional Language*, LNCS, vol. 1595, September 1999, pp. 55–67.

[10] M. Hidalgo-Herrero, *Semánticas formales para un lenguaje funcional paralelo*, Ph.D. thesis, Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2004.

[11] M. Hidalgo-Herrero and Y. Ortega-Mallén, *Continutation Semantics for Parallel Haskell Dialects*, Prog. Lang. & Sys., 1st Asian Symp. Proc. (A. Ohori, ed.), LNCS, vol. 2895, November 2003, pp. 303–321.

[12] J. Hoffmann and Z. Shao, *Automatic Static Cost Analysis for Parallel Programs*, (2013), Submitted Manuscript, Available Online.

[13] A. Johnstone, P. D. Mosses, and E. Scott, *An Agile Approach to Language Modelling and Development*, Innovations in Sys. & Soft. Eng. **6** (2010), 145–153.

[14] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler, *Run Your Research: On the Effectiveness of Lightweight Mechanization*, Proc. ACM SIGPLAN-SIGACT Symp. Princ. Prog. Lang., POPL 2012, ACM, 2012.

[15] H. Krahn, B. Rumpe, and S. Völkel, *MontiCore: a Framework for Compositional Development of Domain Specific Languages*, Int. J. Soft. Tools for Tech. Transfer (STTT) **12** (2010), no. 5, 353–372.

[16] J. Launchbury, *A Natural Semantics for Lazy Evaluation*, Proc. 20th ACM SIGPLAN-SIGACT Symp. Princ. Prog. Lang., ACM, 1993, pp. 144–154.

[17] M. Lumpe, *Growing a Language: The* GLOO *Perspective*, Proc. 8th Int. Conf. Soft. Composition (C. Pautasso and É. Tanter, eds.), LNCS, vol. 4954, Springer, 2008, pp. 1–19.

[18] K. Nakata and M. Hasegawa, *Small-Step and Big-Step Semantics for Call-by-Need*, J. Func. Prog. **19** (2009), 699–722.

[19] N. Nystrom, M. R. Clarkson, and A. C. Myers, *Polyglot: An Extensible Compiler Framework for Java*, Compiler Constr., 12th Int. Conf., LNCS, vol. 2622, Springer-Verlag, April 2003, pp. 138–152.

[20] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 7th ed., McGraw-Hill, 2009.

[21] L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén, *An Operational Semantics for Distributed Lazy Evaluation*, Trends in Func. Prog., vol. 10, Intellect, UK/The U. Chicago Press, USA, June 2009, pp. 65–80.

[22] P. Sestoft, *Deriving a Lazy Abstract Machine*, J. Func. Prog. **7** (1997), no. 3, 231–264.

[23] I. Sommerville, *Software Engineering*, 9th ed., Addison Wesley, 2011.

[24] M. van Eekelen and M. de Mol, *Reflections on Type Theory, λ-Calculus, and the Mind. Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday*, ch. Proving Lazy Folklore with Mixed Lazy/Strict Semantics, pp. 87–101, Radboud U. Nijmegen, 2007.